Avoiding Obsolescence with a Low Cost Scalable Fault-Tolerant Computer Architecture.

Josef Schaff
Naval Air Systems Command
22347 Cedar Point Rd., Unit 6, Code 4.1.11
Building 2185, Suite 3130-B3
Patuxent River, MD 20670, USA

Overview

This new computer architecture can use anything from COTS (Commercial Off-The-Shelf) microcontrollers to the latest high-end processors. It is a distributed fault-tolerant architecture that is dynamically reconfigurable in the event of device failures, and is fully programmable in conventional high level languages. By using a simple two-level hierarchy with redundant control processors that configure the I/O (Input/Output) processor arrangement, even the failure of several processors will have no effect on data. An example is given of a real-time data acquisition system with a total cost for a 16 channel device with mixed sync/async and proprietary baud rates, of less than $500 in parts. This example system can be reconfigured to any arrangement of 16 or less serial interfaces.
The architecture is flexible and can be expanded into two levels: status, health and monitoring; and clustered I/O and processing. Additional expansion to a third level would add adaptive learning aspects. Each processor can be dynamically removed or replaced, and is designed to run a minimal amount of processor-specific software – about 1-2 kilobytes of code, which allows each new type of processor added to be configured to respond as a generic processor / CPU (Central Processing Unit). This facilitates the addition of new processors with a minimal amount of development. Present software may need to be modified to take full advantage of this architecture, although by using currently available distributed processor operating systems, most of the modifications can be avoided. The layout of the architecture allows both obsolete and state-of-the-art processors to work together, and transparent replacement of obsolete processors with newer ones. Some current software design methodologies can be applied to configuring the hardware architecture, such as CORBA – The architecture lends easily to Object Request Brokers - e.g. cluster CPU replacements can be specified by using Interface Definition Language -type description of CPU functionality, making it CORBA-like from a hardware perspective. Further development and acceptance of this architecture can lead to significant cost savings and mitigate obsolescence in future computer design.

Introduction

In general, CPU speeds increase faster than it is practical to replace them following Moore's law - speed doubling every 18 months or so.
Much of current software needs the increased speed for various reasons, which include poor coding practices and inefficiency. There are some of us, of course who yield to the marketing pressures to have the fastest processor commercially available for their own satisfaction. This is something like buying a Ferrari for the sole purpose of driving in funeral processions.
Due to the high cost of constantly upgrading CPUs to the most current, and the financial loss of decommissioning older processors after only two or three years of service, we need to find an effective means of mitigating this built-in obsolescence. The obvious solution would be re-use. There are several ways that we could do this. One would be to completely redesign the CPU's architecture, which would not be in the semiconductor manufacturer's best interest (but neither would effective re-use plans that reduced their future sales volume). Another way would be to design a computer architecture to allow the incorporation of both obsolete and current processors working together and allow future processors to just 'plug-in' to this architecture. We will define our goals for this architecture and details of implementation in the rest of this paper.
Goals:
1) A low-cost, upwardly scalable architecture that is built from current COTS processors. The scalability will allow future processors to work along with obsolete ones in a synergistic way.
2) Full fault-tolerance, where a processor(s) can be physically unplugged without losing data or overall functionality.
We want to do this with the most cost-saving approach. That would mean using obsolete processors in current equipment with state-of-the-art processors added to the architecture without large changes in software or hardware.
The objectives are to produce a seamless scalability between the old processor and new processors, as well as eliminating single point failures with the inherent redundancy of this architecture. Thus, expensive state-of-the-art updates, which rapidly become obsolete, can be replaced with a distributed architecture that supports both

Paper presented at the RTO SCI Symposium on "Strategies to Mitigate Obsolescence in Defense Systems Using Commercial Components", held in Budapest, Hungary, 23-25 October 2000, and published in RTO MP-072.

current, past and future processors working together in a synergistic manner.

The software embedded in each processor is easily maintained code which effectively translates each unique type of processor into a generic one in order to integrate it into this distributed architecture. That also allows each CPU to work with small but powerful real-time operating systems, as well as treat the processors as functional objects to be added or deleted from the distributed architecture.

Current Computing Systems:

In pre-COTS days, the CPU was designed for specific tasks. An example from about 1943 is Colossus, which was an electromechanical processor designed for code-breaking, and had its programming hardwired or set by patch panel jumpers. Later on, software written for processors allowed them to do general tasks and eventually multi-task. More recently, processors were designed for particular classes of problems, such as DSPs for signal processing, 32 / 64 bit CPUs for desktop PCs, and embedded controllers for small and medium scale device control.

We currently have a variety of COTS and proprietary systems that are either networked or stand-alone throughout the world. Typically, COTS life is 2-3 years. Large systems or mission computers, which may be based on COTS components, are usually obsolete by the time that they are fully deployed. This is due to a long (by computer state-of-the-art standards) initial life cycle development and deployment. Advanced proprietary or prototype systems usually have a longer life, but at a much greater cost.

High-end systems with multiple processors and / or special parallel processing schemes use specialized parallel algorithms that tend to lock in software to the specific architecture.

An example laboratory system is shown which replaced a proprietary architecture that had reached the end of its useful or maintainable life. A novel approach was taken to create a small scalable architecture based on COTS that maintained full functionality and most software compatibility with upward expandability.

Example System

This example system was originally a large rack-mounted VME-based system with proprietary boards. Additionally, the base system was obsolete and the proprietary boards had little or no supporting documentation. The objective here was to upgrade this system to a current scalable system for a hardware cost of less than $1000. The system should run a commercially or freely available operating system, and the upgrade should have minimal or no impact on functionality.

The new system should also be readily portable, so it can be designed into a briefcase with a laptop running Linux as a display unit, and a master controller board of about 30x30 CMS in size.

This master board would include the 16 data channels that the original equipment monitored and use four microcontrollers ("Basic Stamp" microcontrollers) to each acquire four channels of serial data at the proprietary baud rates, sync or async depending on channel. This board has the capability to add channels by just adding another microcontroller for each four channels.

(See Fig. 0)

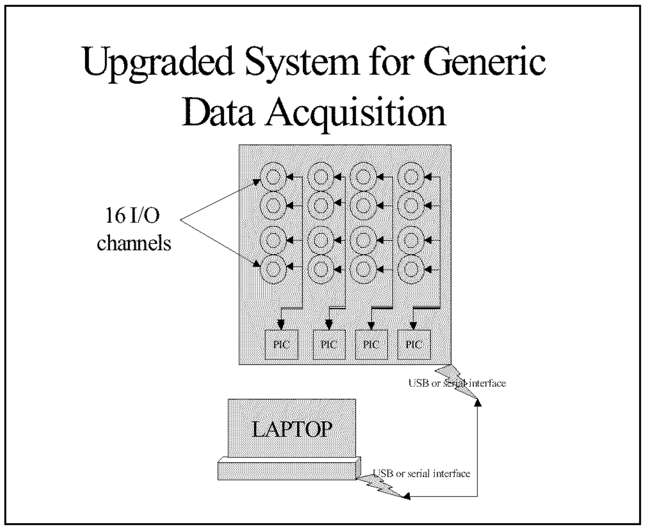


Figure 1 – the generic data acquisition system.

Since each PIC microcontroller chip is less than 4cm², the entire board and connectors is small and fits in the briefcase with the laptop computer. Additionally, each PIC microcontroller has several unused I/O channels that can be used as spares.

In summary:

(1) The large rackmount system was reduced in size to a briefcase.

(2) A rigid, non-expandable system was made upwardly scalable.

(3) The entire system is COTS based.

(4) The system costs less than $1000 in hardware (depending on the cost of the laptop –which could be an older model for $300-$500).

Improving the Original Concept:

Lets re-frame the example's approach by using cheap CPUs in clusters to handle larger problems. Is this like the DIS (Distributed Interactive Simulation), currently renamed HLA (High Level Architecture), where networked systems participate in simulations from any location, and any system may deploy a real-time object into the simulation world such as an adversary or threat platform? Not really, as that is meant to support the HLA's interface aspects for simulations.

We would want something that could handle generic distributed processing as well as specific aspects of computational processing, and perhaps parallel interfacing for multiple data channels. This architecture would be similar in function to the SETI@home web site, where you are offered a chance to participate in SETI's search and data processing as well as a nice screen saver. This is in exchange for allowing your computer to be used during idle time as one node in their distributed processing architecture to process their data. The SETI architecture is primarily a SIMD (Single Instruction Multiple Data) type of parallel machine, where a primary control machine is needed over all the node machines, and this could be a source for single-point failure. A comparison could also be made with our design to the Beowulf architecture, in that it can run on existing hardware, and can use open source software for the most part. There are differences, however in that the Beowulf cluster architecture is designed to run on private high speed LANs, and not over a distributed network. According to the founder of the Beowulf architecture, it will not be designed to run over the Internet, or a similar distributed approach. Again, Beowulf clustering is designed to have a master node control the cluster.

Lets consider the best ways to build an architecture from varied and possibly obsolete components. We would probably want a MIMD (Multiple Instruction Multiple Data) type architecture, or a SIMD / MIMD mix of clusters that can be dynamically reconfigured. In other words, each cluster could be SIMD for fault recovery, and the set of clusters would enable a MIMD architecture. We would also want to fall back to minimal configurations if we should lose many of the clusters. We will examine that first:

Start out with a small architecture, and call it level 0. This architecture may consist of only a few CPUs that have simple rules embedded into about 1-2 kilobytes of code for each processor. The code would also allow each processor to appear generic to the others, such that each one could use a common generic instruction set. There would be no need for a true operating system on each processor. These processors could be microcontrollers, DSPs or other embedded systems as well as high level CPUs. A single operating system would then run over all the processors.

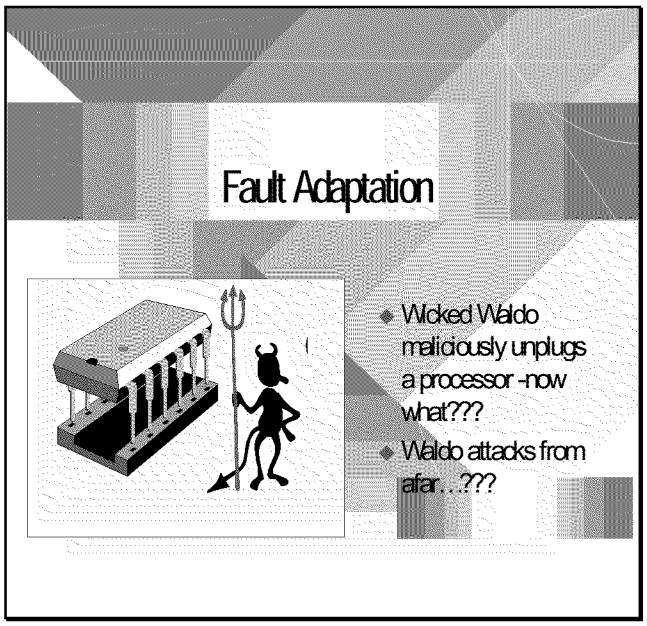
A larger version, which we will call level 1, can use small kernel real-time operating systems for each processor, or just each cluster of processors.

The code embedded in each processor would be similar to the level 0 approach, and may have minor differences for classes of functionality, such as I/O clusters vs. status, health and management clusters. These classes could define clusters as particular types of objects with strong object models defined in tools such as UML, Rhapsody or Rational Rose for object modeling, as well as CORBA extensions.

The operating system running on each processor or cluster would have a small footprint (400k or so) such as in RTMX, PROSE and others, and be POSIX 1003.1b real-time extension compliant.

Fault Adaptation:

If one or more processors are unplugged or damaged, how can we handle this? What if a particular processor has an inherent exploitable vulnerability such that an attack from afar can succeed?



Fault Adaptation

- Wicked Waldo maliciously unplugs a processor -now what???
- Waldo attacks from afar...???

Figure 2 – How do we deal with Faults?

Both these situations call for some type of fault tolerance, usually not found in generic or COTS machines. We can, however determine an architecture that incorporates this scope of fault avoidance.

By using a clustered approach to disparate CPUs we can avoid these shortcomings inherent in conventional systems.

The design described here is optimal for multi-channel I/O intensive operations, but is not limited to that, and has far more diverse uses. An example of this would be low bandwidth but high levels of numerical calculations that work well in a distributed processing environment.

We define three levels of complexity in the architecture (more can be defined later). Each level has processors organized into clusters of two or three for fault avoidance. These clusters can act as a single processor if complete data recovery is mandated, and as such inter-processor communication shares data and processing so that any single processor failure will not affect data or processing capability. In some ways this is similar to the RAID aspect of redundant disks for critical data storage and recovery.

Example Level 0 architecture:
This consists of small clusters of redundant-functionality microcontrollers or standard CPUs that are not necessarily identical, each of which has several I/O channels, with a digital switch layer to isolate any major electrical faults. This way if the external devices that are being interfaced to have noisy data or unstable voltage fluctuations, the processors are not damaged. Since the monitor code executing on each microcontroller is almost identical (identical if same type / model of controller) and consists of a few kilobytes to make these generic in nature, any damage to, or physical removal of any processor does not affect the data or processing in any way. This code also monitors neighbor status and handles basic I/O functions.
We have a small and efficient real-time system, on which we can optionally load a distributed operating system. The operating system would treat each cluster as a single processor, with the embedded monitor code "translating" instructions to run as a generic processor.

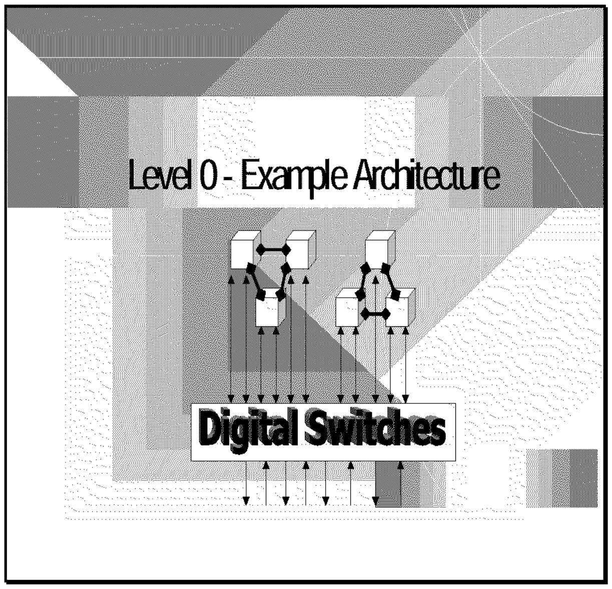


Figure 3 - a level 0 example

Level 0 Rules:

Details on the level 0 rules explain how we can accomplish our objectives for fault management, health monitoring, and generic processor functionality in a small (< 2 Kbytes) code space. For very different processor architectures some parts of this code would need to be modified to accommodate unique aspects. We define five rules that apply to the level 0 architecture, and also to varying degree to the higher levels as well:
Rule 1: *Relation* (intrinsic) - keep related I/O in localized cluster(s). On a small scale this would mean that I/O from a common or related source would be handled by one cluster, so that if a CPU failed, then its alternate(s) would take over for it and request another backup CPU while managing to save the data in temporary storage. By handling faults in this way, a race condition could be avoided which would occur if part of the data were in a non-local CPU when the one in the local cluster failed.
Rule 2: *Association* - direct data to associated destinations (process or memory block in common). This method of "chunking" data also mitigates similar race conditions, or skewed timing problems that were mentioned in rule 1.
Rule 3: *Selection* - "hot" or pre-selected runner-up processor affiliated with active processor in each cluster. This allows a pre-selected replacement for failed CPUs in a lossless manner. In situations where there are an odd number of processors after a fault, the lone CPU would affiliate with either the node of two CPUs under the heaviest load or a node of two CPUs in the closest proximity.
Rule 4: *Health* - IPC (inter-processor communication) or at least "ping" between affiliated processor and runner-up. This keeps a close watch on when a CPU needs replacement due to failure or when a lone CPU can join a cluster by following the *selection* rule above.
Rule 5: *Failure mode* - hunt for available processor if runner-up fails, and check for I/O saturation. Call for "help" from another cluster if saturated. This is the mode that a clustered CPU enters when it loses its associate CPU in the cluster.
We could apply all of this to the example architecture in Figure 1, without any significant increase in cost for hardware.
Level 1:
Inclusive of level 0, but level 1 has functionality divided over two or more layers of clusters - first layer is clustered I/O or CLIO. Next layer(s) is status health and management (SHAM). The original ruleset as well as new rulesets apply, defining more specific boundaries on SHAM and CLIO layers. Enhanced aspects are also applied, such as intelligent / adaptive configuration interfaces, to be used by level 2 architectures.
The functionality still remains overlapping with the level 0 architecture, so that if an entire layer is lost due to failure, the architecture will fall back to level 0 hopefully without any significant loss of data while maintaining full functionality.

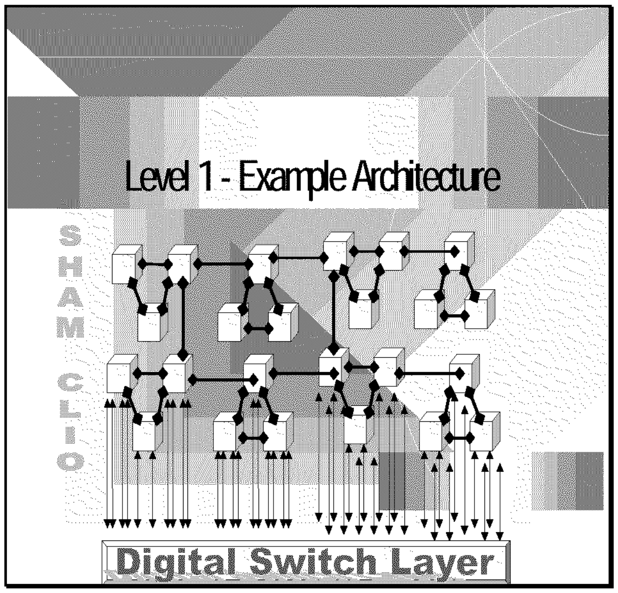


Figure 4 – a level 1 example

Example Level 1 Architecture:
As seen in the diagram, the same clustering is used in the level 1 design, but the clusters that handle I/O are in a separate layer from the status, health and management. The two layers communicate with each other similar to the single layer level 0 via inter-layer channels, and yet maintain the intra-layer and intra-processor communications as well. Inter-layer communications are kept short for the most part, unless a major reorganization of layers needs to take place. This minimizes overhead on interprocessor communication links.
Digital communication in and between layers could be accomplished by an internally incorporated USB interface for 809xx series microcontroller, or could be an Ethernet interface built into a microcontroller (I think somebody already has one out there...). That way each cluster could pick up a portion of the network load processing.
Keep in mind that these processors do not need to be state-of-the-art, but obsolete and inexpensive ones could suffice. Typical standard microcontrollers cost about $1-$2. Older PC CPUs cost $10-$30, and can have up to 50% of the maximum processing power available today.

Fault adaptation on intelligent multi-kernel clusters (Level 2):
This advanced version of the architecture can actively reconfigure itself for fault avoidance, and adapts to hostile attacks. An example would be an attack from a networked intelligent agent that focuses onto perceived weakness in the architecture, or even operating systems executing on it. The system would be able to compensate for and possibly repel future attacks. This is feasible because of the nature of a distributed system like this. If one or more layers handle adaptive learning, then it can behave like a neural network or other adaptive systems. The degree of complexity in the level 0 or level 1 architectures may not be sufficient to accomplish this. However, in this level 2, there are at least three layers, the first two handling the aspects of level 1, and additional layers the adaptive aspects.
Enhancing aspects of the level 2 paradigm can allow separate kernels to run on each node, with socket-based communication handling I/O and IPC.
Several real-time operating systems can exploit the benefits of this architecture. Two examples of operating systems that have excellent security built in are:
1) RTMX - this could run as 1 kernel per node. It exhibits the full Berkeley support for export, NFS and shared memory, and incorporates high level encryption. This has recently been donated to the OpenBSD project, as it is open source code. This means that in future releases of OpenBSD, the real-time portions of RTMX will be incorporated. If these future versions support a small kernel as in RTMX, then OpenBSD can also be a viable operating system for this architecture.
2) PROSE - Developed at Sandia Labs, this could function as 1 kernel per cluster or even layer, as the operating system supports a real-time kernel running over a multi-node network. This was to be certified by NSA to the B3 level.
Both of these can be placed into ROM for each processor or globally shared, as the entire kernel is less than 400 Kbytes in size. They are also both publicly available.

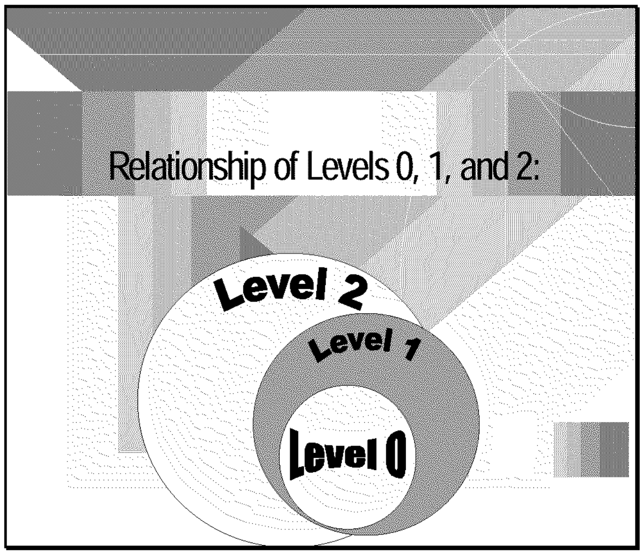


Figure 5 - Relationship between Levels

The relationship between the three levels just described is as subset / superset, where level 0 is a subset of level 1 and level 2, level 1 is for the most part a subset of level 2 with some minor exceptions that have to do with adaptive vs. non-adaptive fault management. The purpose of designing the architecture in this way is to allow full scalability between having a few CPUs form a level 0 to adding on more CPUs to the architecture over time to eventually achieve level 2. Beyond level 2 can still be treated as a level 2 configuration, with enhanced functional features typical of a massively parallel distributed processor machine.

Perspectives and methodologies:
This distributed architecture is similar to a neural network in many ways, not least of which is its ability to adapt and self-organize in larger versions. An interesting aspect that would allow us far more control over the internal organizational interconnectivity would be to use tools from the software engineering world and take an object oriented approach. Currently, the object-oriented approach is applied only to software. For optimal benefits to be derived from object-oriented design, the methodology should be applied system-wide, i.e. to hardware module objects as well as software. We could then bring the hardware and software worlds together into an object based system paradigm.
The architecture described in this paper lends itself to this type of approach. If we treat these architectures as object models then we can use existing tools such as Rhapsody, which is designed to work in embedded systems and is a UML (Unified Modeling Language) visualizing environment with a built-in model checker. This can develop the common ruleset generation for each level, and possibly map layer connectivity.
Hardware layouts can be managed by a CORBA-like environment, with clustered CPU mappings defined by an IDL (interface definition language) and managed by an ORB (object request broker). The objective here is to accomplish a system-wide object-oriented design not just limited to software. Ultimately this may create a more consistent mapping of software processes onto hardware resources.
Finally, and for future research, a large-scale version of this may prove useful as an inexpensive alternative to the quantum computing environment of Shor & Lloyd (Bell Labs / MIT), at least until that becomes economically competitive.

Conclusion:

This distributed architecture is based on COTS systems and essentially does a re-use of obsolescent CPUs. The distributed architecture constructed can be done at minimal cost compared to state-of-the-art, or proprietary systems. It produces a robust architecture that is upwardly scalable, fault tolerant and dynamically reconfigurable so that mission critical data is preserved. The system constructed from this architecture can run real-time and is a distributed parallel computer. Essentially, it is a supercomputer built from obsolete and current components. The trade-off of reliability for extreme speed is done with distributed modularly defined clustering. By organizing the methodology of implementing this architecture into three levels that are based on the degree of functionality and complexity, and basing the core level on a set of intrinsic rules that govern fault related mitigation, we can construct a highly modular paradigm for distributed processing.
The modular design fits well with the concepts of OOD and use of UML for definition, and CORBA aspects such as ORB for hardware modules.
Further development of this architecture can result in defining a new standard to apply to distributed architectures. This standard would simplify hardware redesign through the modularity of an object-oriented hardware paradigm with tremendous cost saving benefits by re-use of existing low cost obsolescent processors.

References:

Quinn, Michael J, “Designing Efficient Parallel Algorithms for Parallel Computers” ©1987, McGraw-Hill Publishing Co.

Beowulf Clusters – Various sources with the best being Scyld Corp: http://www.scyld.com/

PROSE – parallel real-time operating system developed at Sandia Labs:
http://www.cs.sandia.gov/%7Erolf/puma/jrtos/

RTMX – real-time operating system that recently has been donated to OpenBSD organization:
(1) http://www.rtmx.com/
(2) http://www.openbsd.org/

SETI @Home - http://setiathome.ssl.berkeley.edu/